Paper

Breaker: an account agents pay from

An agent key on Solana that pays x402 services from an owner's account, up to a limit the chain enforces.

---

1. Introduction

An agent that pays for what it uses needs something to pay with. Handing it a funded wallet gives it everything in the wallet. Breaker gives it a **budget**: an account the owner funds and limits, which the agent's key spends from as an SPL token delegate. The key holds no tokens and no SOL. The limit is the allowance of the token account itself, set by a Solana program and enforced by the token program on every transfer.

This paper sets out the program's accounts (section 2), the invariant they keep (section 3), how a budget pays an x402 service without any change to x402 (section 4), the reserve behind a panel's budgets and the money around the protocol (section 5), and the design of channels and lines, the next two builds (section 6). Every claim about the program points at a line of its source, `backend/programs/breaker/src/lib.rs`, the build deployed at program `2MSRTKeY8QnmkMrHd5B9EvA9Tb1NerWvL4UpquhjEPet` on mainnet.

2. Accounts

2.1 Config

`Config` (`backend/programs/breaker/src/lib.rs:1469`) holds the admin and the funding fee rate in basis points, `fee_bps` (`backend/programs/breaker/src/lib.rs:1471`). It lives at the seed `["config"]` (`backend/programs/breaker/src/lib.rs:1724`). `init_config` and `set_fee` accept a rate only up to 10,000 (`backend/programs/breaker/src/lib.rs:21`, `backend/programs/breaker/src/lib.rs:36`). On mainnet the rate is 100 basis points.

2.2 Panel

`Panel` (`backend/programs/breaker/src/lib.rs:1477`) is the owner's account: one per wallet, at `["panel", owner]` (`backend/programs/breaker/src/lib.rs:1746`). It stores the owner (`backend/programs/breaker/src/lib.rs:1478`) and a count of open budgets. Every owner instruction finds the panel by the signer's key and checks `has_one = owner` (for example `backend/programs/breaker/src/lib.rs:1844`), so only the owner's signature reaches its money. A budget's settings are reached by the owner's signature or by a member's holding the right role (section 6.3), checked by `authorize` (`backend/programs/breaker/src/lib.rs:1457`).

2.3 Budget

`Budget` (`backend/programs/breaker/src/lib.rs:1485`) lives at `["budget", panel, agent_key]` (`backend/programs/breaker/src/lib.rs:1764`), seeded by the key it was opened with and kept at that address after a key rotation. Its fields carry the terms and the accounting:

Field	Line	Meaning
`agent_key`	`backend/programs/breaker/src/lib.rs:1488`	The current delegate.
`limit`	`backend/programs/breaker/src/lib.rs:1493`	What the agent may move in one period.
`period_seconds`, `period_start`	`backend/programs/breaker/src/lib.rs:1494`	The period, at least 60 seconds (`backend/programs/breaker/src/lib.rs:1589`).
`expiry`	`backend/programs/breaker/src/lib.rs:1496`	An optional end, after which nothing is allowed.
`on`	`backend/programs/breaker/src/lib.rs:1497`	The switch.

Field	Line	Meaning
`spent_at_touch`	`backend/programs/breaker/src/lib.rs:1498`	Spend this period, counted at the last program instruction.
`approved_at_touch`	`backend/programs/breaker/src/lib.rs:1499`	The allowance the program last approved.
`reset_spent_at_touch`	`backend/programs/breaker/src/lib.rs:1503`	The spend at the last reset this period.

2.4 The budget account

The money sits in the budget account: the associated token account of the `Budget` address for the budget's mint (`backend/programs/breaker/src/lib.rs:1767`). Its owner is the `Budget` PDA, so only the program can move it as owner, by signing with the budget's seeds. The agent key is that account's SPL delegate and nothing more. Both SPL Token and Token-2022 mints are accepted; the token program is stored in the budget and checked on every instruction.

2.5 The reserve

The reserve is the associated token account of the `Panel` address for a mint (`backend/programs/breaker/src/lib.rs:1956`). The owner funds it once for the whole panel. A budget set to draw takes what its period's limit calls for from it (section 5). A small `Drawer` account at `["draw", budget]` (`backend/programs/breaker/src/lib.rs:1509`, `backend/programs/breaker/src/lib.rs:1998`) marks a budget as drawing.

3. The invariant

> In any period, an agent key moves out of its budget at most the budget's limit since the period began or since the last reset, and never more than the budget account holds. While the budget is off or expired, it moves nothing.

3.1 The only door

The agent key signs no program instruction that moves money out of the budget account. Its only power over the account is the SPL delegation: the token program lets a delegate move up to `delegated_amount` and lowers that amount by each transfer. A transfer can never exceed the account's balance. So the agent's spend is bounded by the allowances the program grants, and by the balance.

Every other outflow belongs to the owner. `withdraw` sends to the owner's own associated token account (`backend/programs/breaker/src/lib.rs:183`, constrained at `backend/programs/breaker/src/lib.rs:1877`) and requires the amount to be within the balance (`backend/programs/breaker/src/lib.rs:179`). `close_budget` returns the balance to the owner the same way (`backend/programs/breaker/src/lib.rs:389`). `draw` only moves money in (`backend/programs/breaker/src/lib.rs:556`).

3.2 One function grants allowance

Allowance is granted in one place, `set_allowance` (`backend/programs/breaker/src/lib.rs:1605`), through `apply_allowance` (`backend/programs/breaker/src/lib.rs:1618`); only `open_budget`, whose budget is new and has no lines, calls `apply_allowance` directly (`backend/programs/breaker/src/lib.rs:112`). While the budget has lines, `set_allowance` grants zero whatever amount it is given (`backend/programs/breaker/src/lib.rs:1613`, section 6.2). Otherwise it approves the current agent key for an amount, or revokes the delegation when the amount is zero (`backend/programs/breaker/src/lib.rs:1632`, `backend/programs/breaker/src/lib.rs:1641`), signing as the budget PDA, and records the amount in `approved_at_touch` (`backend/programs/breaker/src/lib.rs:1654`).

Before any instruction changes the terms, it accounts for what the agent spent since the last touch. `accrue` (`backend/programs/breaker/src/lib.rs:1516`) computes the spend as the last approval less what is still delegated (`backend/programs/breaker/src/lib.rs:1519`), adds it to `spent_at_touch`, and records the current delegated amount as the new baseline (`backend/programs/breaker/src/lib.rs:1529`). The subtraction is checked, so a delegated amount above the last approval stops the instruction instead of being counted.

What may still be spent is `remaining` (`backend/programs/breaker/src/lib.rs:1535`): the limit less the spend since the last reset, and zero when the budget is off or expired (`backend/programs/breaker/src/lib.rs:1536`). Expiry is `now >= expiry` (`backend/programs/breaker/src/lib.rs:1533`).

3.3 Every path

Each instruction that calls `set_allowance` grants at most `remaining`, so after it `spent_at_touch - reset_spent_at_touch + allowance <= limit`:

Instruction	What it approves	Lines
`open_budget`	`limit`, with spend 0 and the switch on	`backend/programs/breaker/src/lib.rs:75`, `backend/programs/breaker/src/lib.rs:91`, `backend/programs/breaker/src/lib.rs:92`, `backend/programs/breaker/src/lib.rs:112`
`set_limit`	`remaining` under the new terms, after `accrue`	`backend/programs/breaker/src/lib.rs:214`, `backend/programs/breaker/src/lib.rs:218`, `backend/programs/breaker/src/lib.rs:219`
`switch_on`	`remaining`, refused once expired	`backend/programs/breaker/src/lib.rs:254`, `backend/programs/breaker/src/lib.rs:258`, `backend/programs/breaker/src/lib.rs:259`, `backend/programs/breaker/src/lib.rs:260`
`rotate_key`	`remaining`, to the new key; the old key loses the delegation	`backend/programs/breaker/src/lib.rs:275`, `backend/programs/breaker/src/lib.rs:276`, `backend/programs/breaker/src/lib.rs:277`
`reset`	`limit` if on, else 0; it first moves the reset baseline to the current spend	`backend/programs/breaker/src/lib.rs:227`, `backend/programs/breaker/src/lib.rs:231`, `backend/programs/breaker/src/lib.rs:232`, `backend/programs/breaker/src/lib.rs:237`
`roll`	`limit` if on and not expired, else 0, once the period has ended	`backend/programs/breaker/src/lib.rs:308`, `backend/programs/breaker/src/lib.rs:328`, `backend/programs/breaker/src/lib.rs:331`, `backend/programs/breaker/src/lib.rs:336`
`switch_off`	0, with the switch off	`backend/programs/breaker/src/lib.rs:245`, `backend/programs/breaker/src/lib.rs:246`
`close_budget`	0, then returns the balance and closes	`backend/programs/breaker/src/lib.rs:378`, `backend/programs/breaker/src/lib.rs:379`

On a budget with lines every row above grants zero, which keeps the same bound with nothing delegated (section 6.2). The owner's instructions, and a member's (section 6.3), reach these through `touch` and `allow` (`backend/programs/breaker/src/lib.rs:1808`, `backend/programs/breaker/src/lib.rs:1812`), which run `accrue` first.

Within a period, then, the agent's total spend is what was accrued plus what it moves from the current allowance, and each grant keeps that sum at or under the limit counted from the last reset. Raising the limit with `set_limit` grants only the difference that remains under the new limit; lowering it below what

was spent grants nothing (`saturating_sub`, `backend/programs/breaker/src/lib.rs:1538`).

A new period begins only in `roll`, and only after the period's end (`backend/programs/breaker/src/lib.rs:308`). It advances `period_start` by whole periods (`backend/programs/breaker/src/lib.rs:319`), sets the period's spend to zero (`backend/programs/breaker/src/lib.rs:328`, `backend/programs/breaker/src/lib.rs:329`) and grants the limit afresh. `reset` is a deliberate grant of a fresh limit inside a period, by the owner or a Limits member, which is why the invariant counts from it.

While the budget is off, every path grants zero: `remaining` returns zero (`backend/programs/breaker/src/lib.rs:1536`), `reset` grants zero (`backend/programs/breaker/src/lib.rs:232`) and `roll` grants zero (`backend/programs/breaker/src/lib.rs:331`). Only `switch_on` grants again. After expiry, `reset` and `switch_on` refuse (`backend/programs/breaker/src/lib.rs:227`, `backend/programs/breaker/src/lib.rs:254`) and `roll` revokes.

3.4 The reserve keeps the bound

`draw` changes no allowance. On a budget with lines, which holds no delegation, its target is `remaining` instead of the delegated amount (`backend/programs/breaker/src/lib.rs:540`). It moves into the budget account the shortfall between the delegated amount and the balance (`backend/programs/breaker/src/lib.rs:545`), as far as the reserve holds (`backend/programs/breaker/src/lib.rs:546`), and moves nothing when the balance already covers it. So drawing raises the balance toward the allowance and never past it, and the agent's bound stays the allowance. Anyone may call `draw`: the most it can do is bring a drawing budget's account up to what its terms already allow.

4. The payment

4.1 The transaction

When a service answers `402 Payment Required` with an x402 `exact` offer on Solana, `@breaker/sdk` builds a v0 transaction of four instructions (`backend/packages/sdk/src/index.ts:67` to `backend/packages/sdk/src/index.ts:70`):

1. `SetComputeUnitLimit` of 20,000 units.
2. `SetComputeUnitPrice` of 1 micro-lamport.
3. `TransferChecked` from the budget account to the payee's associated token account for the offer's mint, with the agent key as authority, for the offer's amount.
4. A memo, the seller's when it sets one, otherwise 16 random bytes.

The fee payer is the facilitator named in the offer's `extra.feePayer`; the SDK refuses an offer that names the agent key (`backend/packages/sdk/src/index.ts:55`). The agent key signs once (`backend/packages/sdk/src/index.ts:73`), and the SDK never broadcasts: the signed transaction goes to the seller in the `PAYMENT-SIGNATURE` header, and the facilitator adds its fee-payer signature and sends it.

4.2 What the facilitator verifies

The reference facilitator (`@x402/svm` 2.31.0, the exact scheme's `verify`) checks the transaction against the offer:

- the instruction layout and the compute unit limit and price, within its caps;
- that the transfer's authority is not one of the facilitator's own signers, so the fee payer cannot be made to pay;
- that the mint is the offer's asset;
- that the destination is the associated token account of the offer's `payTo` for that mint;
- that the amount equals the offer's amount;
- the memo, when the offer asks for one;
- and that the transaction simulates.

The source account is never compared to anything. Any token account whose authority signs the transfer is accepted, and the token program accepts a delegate as that authority up to its allowance. A budget payment is therefore an ordinary x402 `exact` payment: the seller, the middleware and the facilitator run unchanged. Above the remaining allowance, simulation fails inside the token program and the facilitator answers 402.

4.3 One round trip

A cold payment takes two requests to the service (the request answered 402, then the same request with the payment) and two RPC reads between them: the budget snapshot (`getMultipleAccounts`) and a blockhash (`getLatestBlockhash`). From `@breaker/sdk` 0.2.0, a service the agent has paid before costs one request and, while the snapshot holds, no RPC read:

- After a route's first 402 the SDK keeps its requirements, keyed by method, origin and path (`backend/packages/sdk/src/index.ts:216`, kept at `backend/packages/sdk/src/index.ts:205`), and sends the payment with the next request to that route. Sellers price per route, so the query is left out. A 402 in answer drops what was kept and pays from that 402, so a changed price is paid at the new price.
- The budget snapshot is kept per network and lowered by each payment (`backend/packages/sdk/src/index.ts:168`) until the period it holds ends. Any answer but a 402 counts as paid, since an x402 seller settles before it serves.
- A blockhash is reused for 10 seconds (`backend/packages/sdk/src/index.ts:37`), well inside a mainnet blockhash's life of about 150 slots.
- A remembered price the budget would refuse is not trusted (`backend/packages/sdk/src/index.ts:223`), and every refusal is decided on a fresh read, so the SDK's named errors (`BudgetTripped`, `BudgetOff`, `BudgetExpired`, `InsufficientBalance`, `PaymentTooLarge`) mean what the chain says.

`dev/bench.mjs` runs the research loop of `dev/agent.mjs` against a stand-in x402 service on the local chain, with counters on the SDK's fetch and on its RPC connection. Same budget, same service, 12 paid calls per run, two runs per row:

	0.1.4	0.2.0	0.2.0, a call to a route paid before
Requests to the service per paid call	2.00	1.25	1
RPC reads per paid call	2.00	0.17	0
Time per paid call, 45 ms added to every RPC call and service request	644 and 687 ms	442 and 439 ms	416 and 412 ms

The 45 ms is the measured round trip from the bench machine to its Helius endpoint. On localhost, where every hop costs about a millisecond, time per paid call is the facilitator's settlement in both versions (411 to 433 ms before, 395 to 422 ms after).

5. The reserve and the money

5.1 What draws, and when

`fund_reserve` moves money from the owner into the panel's reserve for a mint (`backend/programs/breaker/src/lib.rs:436`), taking the funding fee on the way in (`backend/programs/breaker/src/lib.rs:438`, `backend/programs/breaker/src/lib.rs:439`). `withdraw_reserve` returns it to the owner, within the reserve's balance, signed by the panel's seeds (`backend/programs/breaker/src/lib.rs:466`, `backend/programs/breaker/src/lib.rs:468`, `backend/programs/breaker/src/lib.rs:473`). `draw_on` and `draw_off` open and close a budget's `Drawer` (`backend/programs/breaker/src/lib.rs:505`, `backend/programs/breaker/src/lib.rs:523`). `open_budget` accepts zero funding, so a budget can be added with a limit and a signature and no money moved.

A drawing budget draws whenever its allowance is set: the API puts `draw` in the same transaction as `reset`, `switch_on` and `set_limit`, opening a drawing budget carries `draw_on` and `draw`, and the keeper sends `roll` and `draw` together at the end of each period. A budget account then holds what its period's limit calls for, and money one agent did not spend stays with the owner in the reserve rather than stranded in a budget.

5.2 The funding fee

Breaker charges one fee: a share of each funding, of a budget or of the reserve. The fee is `floor(amount × fee_bps ÷ 10,000)` (`backend/programs/breaker/src/lib.rs:1600`), computed inside the funding instruction and paid in the same instruction (`backend/programs/breaker/src/lib.rs:1669`); the rest reaches the budget or the reserve. At 100 basis points, funding 250.00 USDC pays a fee of 2.50 USDC and 247.50 USDC reaches the budget. A budget that draws from the reserve pays nothing more, so money is charged once, on its way in. The agent pays nothing, and a payment from a budget arrives at the service whole. Withdrawing, drawing, changing a limit, switching, resetting, rotating a key and closing a budget carry no Breaker fee.

5.3 Where it accrues

The fee accrues to the program's fee account for that mint: the associated token account of the PDA `["protocol"]` (`backend/programs/breaker/src/lib.rs:1773`, `backend/programs/breaker/src/lib.rs:1776`), which no key signs for. For USDC on mainnet it is `9QmAKoHtqTcN9aYS47aiBbj3sB5Y9guibN7yf1NNYbPj`.

5.4 BREAKER and the burn

BREAKER (mint `5s9L2dSDuxAe7QmBcs6GZsy6PN5Wn3Hww3T7LCjBRKR`, supply 1,000,000,000) launched fairly on Raydium LaunchLab, from a launch wallet that does nothing else. The whole supply went to the public launch; the developer's allocation is the developer's own first buy in the launch, on the same terms as every buyer.

LaunchLab pays the launch wallet creator fees from trading in BREAKER. A third of them buys BREAKER on the market, and the wallet burns what it bought; a third of any creator fee paid in BREAKER burns directly. The wallet signs each claim, buy and burn itself, so every one is its own transaction on chain, and Breaker's burn service keeps their ledger by signature. The totals are live at `https://api.breaker.to/v1/token`: SOL claimed and spent in lamports, and BREAKER burned in base units of 6 decimals.

6. The next design

6.1 Channels from a budget

Metered inference and long sessions charge by the stream. On Solana, x402's two streamed schemes, `upto` and `batch-settlement`, settle through one payment-channels program, `CHNLxYvVA28MJP9PrFuDXccuoGXAx7jBacfLEkahyGsX`. Its `open` takes a payer that signs, a rent payer, the payee, the mint, an authorized signer, the channel PDA, the payer's token account and the channel's token account, with arguments `salt`, `deposit`, `grace_period`, `open_slot` and the payout recipients. The channel lives at `["channel", payer, payee, mint, authorized_signer, salt, open_slot]`, and the payer's token account is the payer's associated token account for the mint. The authorized signer signs vouchers for a cumulative amount; the payee settles them (`settle`, `settle_and_seal`), the payer can `request_close`, and `distribute` pays the settled amount to the payout recipients and returns the rest to the payer's token account. `top_up` adds to the deposit and `withdraw_payer` returns the payer's funds to its token account.

A budget fits that program exactly. Its account is the associated token account of the `Budget` address, so with the budget as payer the channel draws its deposit from the budget account, and the program signs as payer by CPI with the budget's seeds, the way it already approves and revokes. In a `batch-settlement` session the agent key is the authorized signer: it signs vouchers and holds nothing, as it does today. In an `upto` call the service's key signs the voucher for what the call used.

Accounts:

- `ChannelRecord` (new, at `["channel", budget, channel]`): the budget, the channel address, the payee, the payout recipient, the authorized signer, the deposit, the part of it counted, the `period_start` it was counted in, and the rent payer.
- `Channels` (new, at `["channels", budget]`): the number of channels the budget has open. It exists only while one is.

Instructions:

- `open_channel(deposit, salt, open_slot, grace_period, recipients)`, signed by the agent key, with the facilitator as rent payer and fee payer. It runs `accrue`, refuses a budget that is off or expired, and requires `deposit <= remaining` and `deposit <= balance`. It adds the deposit to `spent_at_touch`, approves `remaining - deposit` through `set_allowance`, and calls the channel program's `open` with the budget as payer and the budget account as the payer's token account. The channel's payee is the facilitator's key, which opens it; the payout recipient in `recipients` is the service's `payTo`. It records the `ChannelRecord` and adds one to `Channels`. The deposit counts against the period's limit like a payment, so section 3's bound covers delegation and channels together.
- `top_up_channel(amount)`, signed by the agent key: the same accounting for `amount`, then the channel program's `top_up`.
- `distribute_channel(recipients)`, open to anyone: it calls the channel program's `distribute` by CPI. For a sealed channel it reads the deposit and what was settled first, since `distribute`

deallocates a channel past its open window and nothing on chain could say afterwards what it returned. The service is paid, the rest returns to the budget account, and when the record was counted in the current period, `spent_at_touch` falls by the smaller of the refund and the counted part, never below the spend at the last reset, and `remaining` is approved again. A deposit from an earlier period comes back as balance only, since `roll` already began a fresh limit. The record closes to its rent payer, and `Channels` closes with the last open channel. An open channel pays out what is settled and stays open.

- `close_channel`, open to anyone, for a channel distributed outside Breaker: it reads the refund from a channel still in its distributed state, credits it the same way and closes the record. A channel already deallocated credits nothing to the allowance; its money is back in the budget account either way.
- `request_channel_close`, signed by the owner at any time, and open to anyone while the budget is off or expired: it calls the channel program's `request_close` with the budget as payer. After the channel's grace anyone may seal it, and `distribute_channel` returns the rest. The API's switch-off builds add it for each channel the budget has open, and the keeper sends it for a budget that is off or expired with channels open.
- `close_budget` takes the `Channels` address as its one remaining account and refuses while it exists, so a budget closes only once its channels have.

The open reaches Breaker's facilitator the way a payment reaches a facilitator: the SDK builds the `open_channel` transaction, signs it with the agent key, and hands it to the service, whose facilitator adds its fee-payer and rent-payer signature and sends it. A stock facilitator checks an open against one layout, a wallet signing the channel program's `open`, so a service that takes budgets by the stream names Breaker's facilitator: the reference facilitator with Breaker's `open_channel` and `top_up_channel` accepted in that place, every inner account pinned as the reference pins them. A service selling sessions also runs the matching seller scheme, `@breaker/x402-seller`, because the stock one checks the open itself. The indexer follows Breaker's `ChannelChanged` events, and the ledger shows a session as it draws.

6.2 Lines per service

A line names a service a budget pays and how much of the budget's limit that service may take each period. The token program does not restrict where a delegate sends, so a destination is bound on chain only by a channel, whose payout recipient is fixed at open. The channel's payee is the facilitator's key that opened it; the service is the recipient. Lines are therefore channels with a fixed recipient, and a budget on lines holds no delegation at all.

Accounts:

- `Line` (at `["line", budget, recipient]`): the budget, the service's payout recipient (its `payTo`), the cap per period, `spent` in the period `period_start` names, and `committed`, the part of that spend still held in open channels.
- `Lines` (at `["lines", budget]`): the number of lines. While it exists the budget is on lines.

The gate sits where allowance is granted. `set_allowance` is the only function that approves the agent key (section 3.2), and every instruction that calls it now passes the `Lines` address: while that account exists, the grant is zero. So `set_limit`, `reset`, `switch_on`, `rotate_key` and `roll` leave a lined budget's key with no delegation, and an `exact` transfer signed by that key fails in the token program. Money leaves a lined budget's account only through `open_channel` and `top_up_channel`, which move it as the account's owner by CPI and need no delegation.

Instructions:

- `set_line(recipient, cap)`, signed by the owner, with `cap > 0`: it creates the `Line` or sets its cap. The first line needs the budget to have no channel open, creates `Lines` and revokes the delegation; every channel of a lined budget is therefore charged to a line.
- `remove_line(recipient)`, signed by the owner, closes the `Line`. Removing the last one closes `Lines` and approves `remaining` again.
- `open_channel` on a lined budget requires one recipient at the full share, 10,000 basis points, and that recipient's `Line`; `top_up_channel` takes the line of the channel's recipient. The amount must fit the line's cap less its spend this period as well as the budget's `remaining` and balance, and is added to both. A `Line` whose `period_start` is older than the budget's reads its spend as zero the next time it is charged, so a line needs no keeper transaction of its own.
- `distribute_channel` and `close_channel` credit the unspent part back to the line as they do to the budget: only when the deposit was counted in the current period, and never more than the line's `committed`.
- `draw` on a lined budget fills the balance up to `remaining`, since there is no delegated amount to fill to.
- `close_budget` refuses while `Lines` exists; the API's close removes every line in the same transaction.

The bound of section 3 holds for the whole budget, and each line adds its own: in any period, the money moved to a line's recipient is at most that line's cap, and nothing moves to an address without a line. Measured with a leaked key and every way out tried (`dev/leak.mjs`): on a budget with a limit of 1.00 and no lines, the key moves 1.00 to its holder by `exact`. On the same budget with one line of 0.20, it moves nothing to its holder: the `exact` transfer is refused by the token program, a channel paying the holder is refused with `NoLine`, and a channel past the line with `LineOverCap`. A channel to the named service, settled by the holder as payee and voucher signer, moves 0.20, all of it to that service.

A service that charges per call is paid under a line through the same channel: each call is a voucher for the new cumulative amount, settled by the service as it goes. The owner's rule reads directly as accounts: this agent pays search and inference, nothing else, and no more than its line on each.

6.3 Team panels

A company runs its agents from one panel, with more than one person at it. Two things are needed: an owner that is not one person's key, and people who can act on the budgets without holding the money.

The owner can be a multisig with no change to the program. A Squads v4 vault (`SQDS4ep65T869zMMBKyuUq6aD6EgTu8psMjkvj52pCf`) signs as an ordinary signer when one of its vault transactions executes, so a vault opens a panel at `["panel", vault]` and holds every owner power. Each owner action is proposed by a key of the multisig, approved by its keys to the threshold, and executed; the API builds each owner action for a vault-owned panel as that proposal.

A threshold is the wrong shape for stopping an agent: the switch has to work with one press. So the program gains members.

Accounts:

- `Member` (at `["member", panel, key]`): the panel, the member's wallet, and its roles as bits. Bit 1 is **Switch**, bit 2 is **Limits**; a member holds one or both.

Instructions:

- `add_member(key, roles)`, `set_roles(key, roles)` and `remove_member(key)`, signed by the owner, emitting `MemberChanged`. The owner's own key cannot be a member, and roles must be Switch, Limits or both. A removed member's key is refused from the next instruction on.
- The instructions a member may sign take an `authority` signer in the owner's place and an optional `Member`. The program accepts the authority when it is the panel's owner, or when the `Member` at `["member", panel, authority]` holds the instruction's role.
- **Switch**: `switch_off` and `request_channel_close`.
- **Limits**: `set_limit`, `reset`, `switch_on`, `rotate_key`, `open_budget` with zero funding, `draw_on`, `draw_off`, `set_line` and `remove_line`. A member pays the rent of what it opens, and what `draw_off` and `remove_line` close returns its rent to the owner.
- Everything that moves money in or out of a budget or the reserve stays the owner's: `fund`, `withdraw`, `fund_reserve`, `withdraw_reserve`, `close_budget`, a budget's funding at open. So do the members themselves.
- `BudgetChanged`, `LineChanged`, `ChannelChanged` and `DrawChanged` end with `signer`, so every change to a budget names who made it.

Section 3's bound is untouched. A member signs the same instructions with the same grants: every allowance still passes through `set_allowance`, and a Limits member who raises a limit or resets a budget grants exactly what the owner could, out of money the owner put in. A Switch member can only take allowance away. Whoever is on call can switch any budget off with one signature, and can do nothing else.